CSC6210: Advanced multidimensional search — Fall 2009
CSE department, Chinese University of Hong Kong

Lecture 10: Locality sensitive hashing

By Yufei Tao (taoyf@cse.cuhk.edu.hk) Last revision: Nov 9, 2009

Forewords

This lecture discusses *nearest neighbor (NN) search* in high-dimensional space. Specifically, let $\mathcal{D}$ be a set of d-dimensional points, where d is a large number. Given a query point q, the goal of NN search is to return the point $o^* \in \mathcal{D}$ that is the closest to q among all points in $\mathcal{D}$. Formally, for any point $o \in \mathcal{D}$, it holds that $\|o^*, q\| \leq \|o, q\|$, where $\|.,.\|$ denotes the distance of two points.

NN search has been very well solved when the dimensionality d is small. For example, a well-known index structure called the *R-tree* [1, 4] (which we will study later in this course) can be used to answer NN queries efficiently for up to $d = 5$. There exist other structures (e.g., the *iDistance* [7]) that can handle higher dimensionalities. However, all these structures are eventually outperformed by a simple *brute-force scan* when the dimensionality reaches a certain level. This frustrating phenomenon is known as the *curse of the dimensionality*.

Fortunately, many practical applications do not require *exact* answers to NN queries. Instead, they can tolerate some small imprecision, that is, it is acceptable to return an *approximate NN*, which is not too far from the query than its real NN. This motivates the definition of *c-approximate NN*. Specifically, a data point $o \in \mathcal{D}$ is a *c-approximate NN* of a query q if its distance to q is at most c times the distance from q to its real NN o^*, namely, $\|o, q\| \leq c\|o^*, q\|$. The constant c is referred to as the *approximation ratio*.

Several effective techniques have been proposed to perform c-approximate NN search. In this lecture, we will study an important technique called *locality sensitive hashing* (LSH) [6]. A salient feature of LSH is that it works well in *arbitrarily high dimensionality* d. In particular, regardless of the value of d, LSH always guarantees c-approximate results with good space and query complexities. Furthermore, it is fairly simple to implement and can be easily incorporated in a relational database, even though its underlying theory is a somewhat complex.

To facilitate discussion, we assume that the distance metric is the ℓ_2 norm, but LSH can be adapted to support many other distance metrics as well. Furthermore, we assume that there is a lower bound of 1 on the distance of two points that do not coincide with each other. This is usually true in practice with proper scaling.

1 Ball cover

LSH does not solve c-approximate NN queries directly. Instead, it is designed [6] for a different problem called *c-approximate ball cover* (BC). Let $\mathcal{D}$ be a set of points in d-dimensional space. Denote by $B(q, r)$ a ball that centers at the query point q and has radius r. A c-approximate BC query returns the following result:

(1) If $B(q, r)$ covers at least one point in $\mathcal{D}$, return a point whose distance to q is at most cr.

(2) If $B(q, cr)$ covers no point in $\mathcal{D}$, return nothing.

(3) Otherwise, the result is undefined.

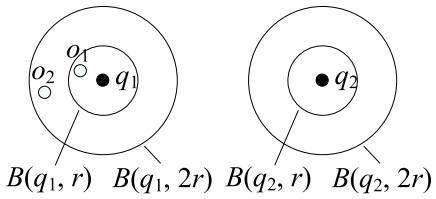

Figure 1: Illustration of ball cover queries

Figure 1 shows an example where $\mathcal{D}$ has two points o_1 and o_2. Consider first the 2-approximate BC query q_1 (the left black point). The two circles centering at q_1 represent balls $B(q_1, r)$ and $B(q_1, 2r)$ respectively. Since $B(q_1, r)$ covers a data point o_1, the query will have to return a point, but it can be either o_1 or o_2, as both of them fall in $B(q_1, 2r)$. Now, consider the 2-approximate BC query q_2. Since $B(q_2, 2r)$ does not cover any data point, the query must return empty.

2 Locality-sensitive hasing

Let $h(o)$ be a hash function that maps a d-dimensional point o to a one-dimensional value. It is *locality sensitive* if the chance of mapping two points o_1, o_2 to the same value grows as their distance $\|o_1, o_2\|$ decreases. Formally:

Definition 1. Given a distance r, approximation ratio c, probability values p_1 and p_2 such that $p_1 > p_2$, a hash function $h(.)$ is (r, cr, p_1, p_2) *locality sensitive* if it satisfies both conditions below:

1. If $\|o_1, o_2\| \leq r$, then $Pr[h(o_1) = h(o_2)] \geq p_1$;
2. If $\|o_1, o_2\| > cr$, then $Pr[h(o_1) = h(o_2)] \leq p_2$.

□

LSH functions are known for many distance metrics. For ℓ_2 norm, a popular LSH function is defined as follows [2]:

$$h(o) = \left\lfloor \frac{\vec{a} \cdot \vec{o} + b}{4} \right\rfloor. \tag{1}$$

Here, $\vec{o}$ represents the d-dimensional vector representation of a point o; $\vec{a}$ is another d-dimensional vector where each component is drawn independently from the normal distribution [2]; $\vec{a} \cdot \vec{o}$ denotes the dot product of these two vectors. Finally, b is uniformly drawn from $[0, 4)$.

Equation 1 has a simple geometric interpretation. Assuming dimensionality $d = 2$, Figure 2 shows the line that crosses the origin, and its slope coincides with the direction of $\vec{a}$. For convenience, assume that $\vec{a}$ has a unit norm, so that the dot product $\vec{a} \cdot \vec{o}$ is the projection of point o onto line $\vec{a}$, namely, point A in the figure. The effect of $\vec{a} \cdot \vec{o} + b$ is to shift A by a distance b (along the line) to a point B. Finally, imagine we partition the line into intervals with length 4; then, the hash value $h(o)$ is the ID of the interval covering B.

The intuition behind such a hash function is that, if two points are close to each other, then with high probability their shifted projections (on line $\vec{a}$) will fall in the same interval. On the other hand, two faraway points are very likely to be projected into different intervals. The following is proved in [2]:

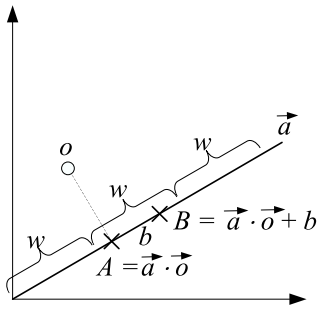

Figure 2: Geometric interpretation of LSH

Lemma 1. Equation 1 is $(1, c, p_1, p_2)$ locality sensitive, where p_1 and p_2 are two constants satisfying $\frac{\ln 1/p_1}{\ln 1/p_2} \leq \frac{1}{c}$. □

3 The structure

Next, we describe an index structure for solving the ball cover problem (see Section 1). As we will see, the structure preserves the spatial proximity of the underlying dataset $\mathcal{D}$, but in a way very different from conventional multidimensional structures (e.g., the R-tree, iDistance, and so on). Without loss of generality, assume $r = 1$; for $r = r' > 1$, the problem can be converted to the case $r = 1$ by shrinking the data domain by a factor of r'.

Denote by $\mathcal{H}$ the *family* of hash functions in the form of Equation 1. Namely, for any combination of $\vec{a}$ and b, $\mathcal{H}$ contains a function that is defined by $\vec{a}$ and b. Note that all the member functions in $\mathcal{H}$ share the same r, c, p_1, and p_2 whose meanings are described in Definition 1. Furthermore, as indicated in Lemma 1, the value of r equals 1.

To build an index structure on the dataset $\mathcal{D}$, we choose independently m random members $h_1(.)$, $h_2(.)$, ..., $h_m(.)$ of $\mathcal{H}$, where m is given by

$$m = \log_{1/p_2} n \tag{2}$$

with n being the cardinality of the dataset $\mathcal{D}$. For each data point $o \in \mathcal{D}$, each $h_i(o)$, $1 \leq i \leq m$, converts o to a 1D value. Hence, by concatenating all the m values together, we obtain an m-dimensional hash vector $G(o)$:

$$G(o) = \langle h_1(o), h_2(o), ..., h_m(o) \rangle. \tag{3}$$

If all the points $o \in \mathcal{D}$ with the same $G(o)$ are grouped into a *bucket*, essentially we have created a hash structure on $\mathcal{D}$, using $G(.)$ as the hash function (which in turn composites m independent hash functions).

If two points o_1, o_2 are far away from each other, they are unlikely to fall in the same bucket. In particular, since each $h_i(.)$ is $(1, c, p_1, p_2)$ locality sensitive, the probability of $h_i(o_1) = h_i(o_2)$ is at most p_2 for any two points o_1 and o_2 whose distance is larger than c. The independence of $h_1(.)$, $h_2(.)$, ..., $h_m(.)$ then ensures that the probability of $G(o_1) = G(o_2)$ is at most p_2^m. By choosing a sufficiently large m, p_2^m can be made arbitrarily small. Our choice in Equation 2 makes $p_2^m = 1/n$, which allows us to derive the nice quality guarantee of LSH, as elaborated in Section 5.

Preventing distant points from falling in the same bucket is crucial for preserving the spatial proximity, but it alone is not enough. It is equally important to ensure that *close* points will appear in the same bucket with high probability. Unfortunately, this cannot be accomplished using a single hash structure. We could claim that (similar to deriving p_2^m) if two points o_1, o_2 are within distance 1, they fall in the same bucket with probability at least p_1^m. This, however, is not useful because p_1^m is not a large probability given the value of m in Equation 2. In general, we have two conflicting goals here — p_2^m needs to be small, but conversely, p_1^m has to be large. Unfortunately, the p_1 and p_2 of Equation 1 differ only slightly, such that no m fulfills both purposes at the same time.

LSH circumvents the problem cleverly by building multiple hash structures. Specifically, by repeating the construction procedure described earlier independently l times, we create l hash structures T_1, T_2, ..., T_l, where l is given by:

$$l = n^{1/c}. \tag{4}$$

As explained in detail later, such an l ensures that if o_1 and o_2 are close, then they fall in the same bucket in *at least one* hash structure with high probability.

Each hash structure occupies the same space $O(nd)$ as the dataset $\mathcal{D}$ (note that $O(d)$ space is needed to store a point's coordinates). Hence, the total space is $O(dn^{1+1/c})$.

4 Query algorithm

As explained in Section 3, the index consists of l independent hash structures T_1, T_2, ..., T_l. Recall that each structure T_i ($1 \leq i \leq l$) is built from a (composite) hash function G_i as illustrated in Equation 3. Next, we discuss how to answer a NN query q.

First, the query algorithm identifies the bucket I_i in each T_i that q is hashed to, i.e., I_i corresponds to $G_i(q)$. Then, the algorithm simply probes all the data points in the l buckets I_1, I_2, ..., I_l, if they totally contain at most $2l + 1$ points (including duplicates). In case the total size of the l buckets is greater than $2l + 1$, then the algorithm simply probes $2l + 1$ arbitrary points in those buckets (it does not matter how many points are probed from each bucket). Finally, from the points probed, the algorithm identifies the one o that is nearest to q. If $\|o, q\| \leq c$, o is returned; otherwise, the algorithm returns nothing.

Since at most $O(l)$ points are processed, the query time is $O(dl) = O(dn^{1/c})$, noticing that evaluating the ℓ_2 distance of a point takes $O(d)$ time. Note that the query time is significantly lower than the cost $O(nd)$ of brute-force scan.

5 Analysis of quality guarantee

In this section, we prove that, with high probability, the query algorithm in Section 4 returns a correct result for c-approximate ball cover (where $r = 1$). For convenience, we say that a point $o \in D$ is *close* (to the query q) if it is covered by ball $B(q, 1)$ (which centers at q and has radius 1). On the other hand, we say that a point $o \in D$ is *far* if it is outside ball $B(q, c)$.

We will show that if ball $B(q, 1)$ covers a point o^* in the dataset $\mathcal{D}$, then the algorithm returns a point in $B(q, c)$ (rigorously speaking, it is also necessary to show that if $B(q, c)$ is empty, then nothing is returned; but this is trivial). Observe that this is correct if both of the following properties hold:

P1. At most $2l$ far points are in the same bucket as q in all the hash structures.

P2. o^* falls in the same bucket as q in at least one hash structure.

We have the following crucial lemma.

Lemma 2. Properties **P1** and **P2** hold simultaneously with at least constant probability.

Proof. A far point has probability at most $p_2^m = 1/n$ of falling in the same bucket as q in *one* hash structure. Hence, the expected number of far points in the same bucket as q in l hash structures equals l. By the Markov Inequality, the probability that there are more than $2l$ such far points is at most $1/2$. Namely, the probability that property **P1** does not hold is at most $1/2$.

In a hash structure, point o^* has probability at least p_1^m to fall in the same bucket as q. Hence, the probability that o^* is not in the same bucket as q in *any* of the l hash structures is at most $(1-p_1^m)^l \leq (e^{-p_1^m})^l$, which is at most $1/e$ with our choices of m and l. In other words, the probability that property **P2** does not hold is at most $1/e$.

By the Union Bound, the probability that at least one of **P1** and **P2** does not hold is at most $1/2 + 1/e$. Hence, both properties hold with probability at least $1/2 - 1/e > 0.13$. □

Remark. As a standard trick, by increasing l by a constant factor of $\log(1/\delta)$, the success probability can be boosted to at least $1 - \delta$, for arbitrarily small δ.

6 From ball cover to nearest neighbor search

We have shown that LSH can be used to answer approximate ball cover queries. Interestingly, approximate NN retrieval can be reduced to approximate ball cover search, which enables LSH to be applied for NN search. The simplest reduction works as follows [3]. We can create an index to support $\sqrt{c}$-approximate ball cover with radius $r = 1, \sqrt{c}, (\sqrt{c})^2, (\sqrt{c})^3, ...$, respectively. Given a query point q, we execute a ball-cover query on these indexes in ascending order of their r, until the search at any index returns a non-empty result, which is directly returned as the final result. It is not hard to verify that such a result is a c-approximate NN of q. The drawback of this approach is that a large number of indexes must be stored and querying, thus incurring expensive space and query cost. In the next lecture, we will discuss another technique to overcome the drawback.

Finally, it is worth mentioning that there exist complicated NN-to-BC reductions [5, 6] that have better space and query complexities. However, those reductions are highly theoretical, and are difficult to implement in practice.

References

[1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of ACM Management of Data (SIGMOD)*, pages 322–331, 1990.

[2] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry (SoCG)*, pages 253–262, 2004.

[3] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. of Very Large Data Bases (VLDB)*, pages 518–529, 1999.

[4] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. of ACM Management of Data (SIGMOD)*, pages 47–57, 1984.

[5] S. Har-Peled. A replacement for voronoi diagrams of near linear size. In *Symposium on Foundations of Computer Science (FOCS)*, pages 94–103, 2001.

[6] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. of ACM Symposium on Theory of Computing (STOC)*, pages 604–613, 1998.

[7] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems (TODS)*, 30(2):364–397, 2005.